JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2009; **233**:21–4 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Optimizing Header File Include Directives

*† Diomidis Spinellis[1]

[1] *Athens University of Economics and Business, Patision 76, GR-104 34 Athens, Greece.*
Email: dds@aub.gr

SUMMARY

A number of widely used programming languages use lexically included files as a way to share and encapsulate declarations, definitions, code, and data. As the code evolves files included in a compilation unit are often no longer required, yet locating and removing them is a haphazard operation, which is therefore neglected. The difficulty of reasoning about included files stems primarily from the fact that the definition and use of macros complicates the notions of scope and of identifier boundaries. By defining four successively refined identifier equivalence classes we can accurately derive dependencies between identifiers. A mapping of those dependencies on a relationship graph between included files can then be used to determine included files that are not required in a given compilation unit and can be safely removed. We validate our approach through a number of experiments on numerous large production-systems.

KEY WORDS: C, C++, header files, include directive, preprocessor

1. Introduction

A notable and widely used [1] feature of the C, C++, and Cyclone [2] programming languages is a textual preprocessing step performed before the actual compilation. This step performs *macro substitutions* replacing, at a purely lexical level, token sequences with other token sequences, *conditional compilation*, *comment removal*, and *file inclusion* [3, §3.8]. As program code evolves, elements of it may no longer be used and should normally be pruned away through a refactoring

*Diomidis Spinellis. Optimizing header file include directives. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(4):233–251, July/August 2009. (doi:10.1002/smr.369)
†This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.
Contract/grant sponsor: European Community Sixth Framework Programme: Software Quality Observatory for Open Source Software (SQO-OSS); contract/grant number: IST-2005-033331

[4, 5, 6] operation. Detecting unused functions and variables is a relatively easy operation: the scope where the given element appears is examined to locate references to it. Many compilers will issue warnings for unused elements appearing in a given file or block scope; detecting unused elements in identifiers with external linkage is a simple matter of processing definition and reference pairs of the files to be linked.

A more difficult and also important task is the detection of header files that are needlessly included in a compilation unit. The task is difficult, because macros complicate the notion of scope and the notion of an identifier [7, 8, 9]. For one, preprocessor macros and file inclusion can extend the scope of C-proper identifiers. This is for example the case when a single textual macro using a field name that is incidentally identical between two structures that are not otherwise related is applied on variables of those structures. This implementation pattern is often used to implement via the C preprocessor structural subtyping (in C++ this is achieved using the template mechanism). In addition, new identifiers can be formed at compile time via the preprocessor's concatenation operator. It is therefore difficult to determine if identifiers appearing in an included file are used within the main body of a compilation unit or other subsequently included files.

In the remainder of this section we will discuss why removing unneeded headers is important and also the context of our work. Subsequent sections describe the approach we propose for dealing with the problem (Section 2), its validation (Section 3), and possible extensions (Section 4).

1.1. Motivation

The detection and removal of needlessly included files is important, for a number of reasons.

Namespace Pollution An included header file is a larger and more unstructured element than a single function or variable. The included file can contain code, data, macro definitions, and other recursively included files. All these pollute the identifier namespace, and can therefore affect the compilation of subsequent code, sometimes resulting in subtle and difficult to locate compile or even run–time errors. Table I documents the breakdown of the various identifiers occurring in header files for six large software code bases.[‡] Note that the namespace pollution manifests itself both when a header file's identifiers appear in different roles in subsequently processed code, and when code previously processed (typically through the inclusion of another header file) contains identifiers that clash with those defined in a subsequently included header file. Due to the global visibility of preprocessor elements, a macro can interfere even with identifiers whose scope is a single function block or an individual structure.

Spurious Dependencies The compilation of C code is typically performed by a tool like *make* [10] that (re)builds object files based on their dependencies. Makefiles often contain an (automatically constructed) section identifying the header dependencies for every compilation unit. Consequently, if a file includes headers that it does not require, it will get compiled more often, thus increasing the build effort.

[‡]Details of each system appear in Section 3.

Table I. Header file characteristics.

Metric	FreeBSD	Linux	Solaris	WRK	PostgreSQL	GDB
LOC (thousands)	3,867	3,431	2,951	829	578	362
Header files	5,206	2,506	1,840	228	321	419
Include directives	38,278	45,564	28,788	642	1,196	3,367
... of which unneeded	2,101	865	861	26	17	78
... in f.p. files (Sec. 3.2)	759	339	366	8	12	38
Average number per header file:						
Lines	353	242	287	1,009	202	208
Identifiers	47.8	111.1	122.7	301.7	83.2	85.4
Of which:						
Local to header file	23.3	43.7	56.9	218.5	64.8	61.1
Macros	20.6	41.5	43.0	75.7	19.4	21.2
Typedefs	1.2	2.2	4.9	23.8	3.3	1.8
Structure or union tags	1.5	3.2	4.4	9.8	2.0	1.7
Structure or union fields	11.6	28.1	33.7	62.8	12.2	10.2
Enumeration constants	1.2	4.4	1.3	10.0	2.2	5.6
File-scoped objects	2.9	10.8	6.4	33.9	5.5	7.5
Global-scoped objects	3.0	7.4	14.4	32.9	20.7	20.2

Compilation Time Removing included header files reduces the code that the compiler must process, and should therefore reduce a project's compilation time. Although in our test cases we have found this effect to be negligible (a decrease of compilation time below 5%), there may be projects where these savings are significant.

1.2. Work Context

To the best of our knowledge this paper contains the first description of an efficient generic algorithm for optimizing include file directives. Similar functionality seems to be provided by Klockwork,[§] a commercial tool, which according to its vendor "provides architectural analysis to identify violations such as the number of times a header file can be included." The theory behind the operation of Klockwork isn't publicly known. However, its published interface shows that Klockwork will identify: extra includes (the files also identified by our approach), missing includes with a context dependency, missing includes with a transitive dependency, and files that are not self-compilable.

[§] `http://www.klocwork.com/products/k7_architecture.asp`

In addition, the FreeBSD operating system distribution includes a shell script, named *kerninclude.sh*,[¶] which detects in the FreeBSD kernel source tree include statements that are not required. While the script is project and compiler–specific its approach could be applied to other systems. The script employs a clever brute force algorithm. It works by first locating the include directives in each source file. It then compiles a pristine copy of the source file, as well as modified versions where a single include directive is temporarily removed. Finally, it verifies that the file can be permanently removed through a number of steps.

- It tries to see that the file can be compiled,
- if it can, it compares the resultant object file with that of the pristine version,
- it checks that the removed header is not also included through a directive nested in another file,
- it verifies that no additional compiler warnings were issued, and
- it ensures that the included file is not in a conditional compilation block.

An advantage of this approach is that, for the configurations tested, the correctness of the obtained results is self-evident. At the time of writing the *kerninclude* tool had not been for six years, and it was therefore not possible to run it and obtain empirical results of its performance. However, the computational cost of this approach is intrinsically high, because the number of times it processes each file is equal to the number of include directives in it, multiplied by the number of different software configurations. As we shall see in Section 2.3, the corresponding cost of our approach depends only on the number of configurations. Also, this method will silently fail in cases where timestamps are embedded in object files (for instance through the used of the `__TIME__` predefined macro). Finally, the approach depends on the existence of a complete cross-compilation tool chain for processing non-native software configurations.

Other related work in our area, does not cover the problem we are addressing, but advances the state of the art in the building blocks we use for putting together the proposed solution. Specifically, such work covers the analysis of C code containing preprocessor directives, the removal of dead code, and the handling of multiple configurations implemented through conditional compilation directives.

On the preprocessor analysis front the main approach involves creating a two-way mapping between preprocessor tokens and C-proper identifiers. This was first suggested by Livadas and Small [11], and subsequently used as a way to refactor C code [9, 12]. A recent paper has proposed a GXL [13] schema for representing either a static or a dynamic view of preprocessor directives [14]. Our approach is based on our earlier work [9], which, compared to the method described in reference [12], has the advantage of handling tokens generated at compile time through C's token concatenation operator.

Our system removes dead code from compilation units. Related approaches include graph-based analysis of program elements [15], and the partial evaluation of conditional compilation blocks in order to remove unwanted legacy configurations [16]. The approach described in reference [15] processes file elements in a finer granularity refactoring their elements to minimize unrelated dependencies and thereby speed up the build process. This type of refactoring is more aggressive than what we propose. As a result it can obtain a measurable efficiency improvement on the build process, but at the cost of more invasive changes to the code. The work by Baxter and Mehlich [16] addresses a different problem:

[¶] `http://www.freebsd.org/cgi/cvsweb.cgi/src/tools/tools/kerninclude/`

that of dead legacy code residing in conditionally compiled blocks. Through the partial evaluation of preprocessor conditionals such blocks can be identified and removed. Their work complements ours, because it allows additional code elements to be removed. A key difference is that their approach requires the manual identification and specification of macros defining unwanted configurations.

The handling of multiple configurations implemented through preprocessor directives has also been studied in other contexts, such as the type checking of conditionally compiled code [17] and the use of symbolic execution to determine the conditions associated with particular lines of code [18]. Again, these papers solve different problems, but indicate the high level of research interest in the area of the interactions between the preprocessor and the language proper.

2. Approach Description

Our strategy for locating files that need not be included in a given compilation unit involves three distinct tactics.

1. The establishment and use of a theory for determining when two identifiers are semantically related in the face of the scope distortion introduced by the preprocessor.
2. The processing of individual compilation units (possibly multiple times to take into account conditional compilation) marking definition-reference relationships between files according to the established identifier relationship rules.
3. The postprocessing of the above data to divide the included files into those required for compiling a given unit and those not required.

In this section we describe the application of these tactics in ISO C programs; a similar strategy can be applied to C++ code.

2.1. Identifier Scope in the Presence of the Preprocessor

In order to establish dependency relationships between included files, we need to determine when two identifiers are related. When these participate in a definition-reference relationship that spans a file boundary, this indicates that the file where the identifier is defined needs to be included by the file where the identifier is referenced. Identifier equivalence in the presence of preprocessing involves tracking four types of identifier relationships: semantic, lexical, partial lexical, and preprocessor equivalence.

The most straightforward type of identifier equivalence is *semantic equivalence*. We define two identifiers to be semantically equivalent if these have the same name and statically refer to the same entity following the language's scoping and namespace rules. In the example below the two instances of *errno* are semantically equivalent.

```
extern int errno;
...
printf ("\%s", strerror (errno));
```

Furthermore, by taking into account the scope and semantics of preprocessing commands we can establish that two tokens are *lexically equivalent*: changing one of them would require changing the other for the program to remain correct. In the following example the three instances of the *len* structure

member are lexically equivalent. If the structure and macro definitions occurred in three separate header files, all three would have to be included for the assignment to compile.

```
struct Wall { int len; };
struct Window { int len; };
#define length(x) (x)->len
...
struct Wall * wall_ptr ;
struct Window *window_ptr;
int d = length( wall_ptr ) - length(window_ptr)
```

Moreover, when new identifiers are created at compile time, through the preprocessor's token concatenation feature, *partial lexical equivalence* is used to describe how parts of an identifier can be equivalent to (parts of) another identifier. As an example, in the following code the variable *sysctl_reboot* for the purposes of determining equivalence consists of two tokens: *sysctl_* (which is equivalent to the part also appearing in the macro body) and *reboot* (which is equivalent to the argument of the *sysvar* macro invocation).

```
#define sysvar(x) volatile int sysctl_ ## x
sysvar(reboot);
...
        if ( sysctl_reboot )
```

Finally, *preprocessor equivalence* is used to describe token relationships resulting purely from the semantics of the preprocessing. Thus, in the example below the two instances of *PI* are equivalent and indicate a defintion-reference relationship between the files they occur in.

```
#define PI 3.1415927
double area = PI * r * r;
```

The theoretical underpinning and detailed description of algorithms for establishing the above equivalence classes can be found in reference [9]. In a summary the core algorithm involves splitting the original non-preprocessed source code into tokens, and assigning each one to a unique equivalence class. The code is then preprocessed with tokens resulting from macro-expansion maintaining references to the tokens from which they were derived. During preprocessing, when two tokens match under the rules of *preprocessor equivalence* we described, the corresponding equivalence classes are merged into one. The code is then parsed and semantically analyzed. When two identifier tokens belonging to different equivalence classes are found to be equivalent under the rules of *lexical equivalence* (for instance a variable name in an expression matches its definition) then the two separate equivalence classes are also merged into one. When one or both identifiers consist of multiple preprocessor-concatenated tokens their equivalence classes are first cloned to cover token parts of equal length, and then the paired parts are merged. This last case covers *partial lexical equivalence*. Each merging of equivalence classes allows us to identify an instance where an identifier depends on another and thereby establish dependencies between files.

The implementation of our approach depends on the close collaboration of a standard C preprocessor with what amounts to the front-end of a C compiler. For our approach to work on real-life code the C preprocessor has to handle the many tricky preprocessor uses, such as recursive macro expansion.

Details regarding the implementation of this part can be found in another article [19]. The idea behind the algorithm is to expand as many macros as possible, as long as there is no danger of falling into an infinite recursion trap. The algorithm uses the notion of a hide set associated with each token to decide whether to expand the token or not. Initially, each token starts with an empty hide set, but during macro expansion the tokens accrue in their hide sets the macros that were used during the expansion. The recursive expansion of macros with a hide set obtained through the intersection of the hide sets of the tokens involved achieves the greatest amount of macro replacement without entering into an infinite loop.

2.2. File Relationships

Having established the ways identifiers can depend on each other, our problem is to determine the set of files that were needlessly included in a given compilation unit. For this purpose the relationships between the files participating in the processing of the compilation unit can be established by creating and processing:

- a set of unique file identifiers, F, and
- three binary relations from file identifiers to sets of file identifiers.

To avoid complications arising from referring to the same file through different directory paths or through filesystem links the file identifiers should uniquely correspond to each file irrespective of its path and name; most operating systems can provide this functionality. For instance, on Unix systems a unique identifier is the pair (*inode*, *device-number*), while Windows systems provide an API for transforming an arbitrary file path to a file path that uniquely identifies the corresponding file.

The relations we maintain for each compilation unit are the following.

Providers The *providers* relation R_p maps a compilation unit $c \in F$ into the set of files $R_p(c)$ that contribute code or data to it.

Includers The *includers* relation R_i maps every file $f \in F$ participating in the compilation of unit c onto the set of files $R_i(c, f)$ that include it directly.

Definers The *definers* relation R_d maps every file $f \in F$ participating in the compilation of unit c onto the set of files $R_d(c, f)$ containing definitions needed by it.

The right-hand-side of the *providers* relation R_p for a compilation unit c being processed is easily established by starting with the empty set, and then adding to it each and every file $f \in F$ that contributes to c code (function definitions or statements) or data (variable definitions, or initialization data).

$$\begin{aligned} R_p(c) &= \varnothing \\ R'_p(c) &= R_p(c) \cup \{f\} \end{aligned}$$

We consider data as a special case, because initialization values are sometimes automatically generated and read into a compilation unit's initializer from a separately included file that only contains data. Following our definition for *providers*, files that only contain preprocessor directives and declarations (such as most library header files) are not providers.

The *includers* relation is also easily established by starting with an empty set, and updating it for every instance of an *include* directive taking into account the file $d \in F$ containing the *include* directive and the file $i \in F$ being included.

$$\begin{aligned} \forall f \in F : R_i(c, f) &= \varnothing \\ R'_i(c, i) &= R_i(c, i) \cup \{d\} \end{aligned}$$

The updating of the *definers* relation R_d is more complex, and involves taking into account the equivalence classes presented in the previous section. Again, the relation's base-case value is the empty set.

$$\forall f \in F : R_d(c, f) = \varnothing$$

Each equivalence class can be said to be *rooted* on a definition (or declaration) of one of the identifiers that are its members. Subsequently encountered equivalent identifiers are *references* to the original definition. Although there are cases where the same identifier can be legally defined in the same scope multiple times (two representative examples are *common* variable definitions and macro redefinitions with the same body) these can in practice be safely ignored. Thus, whenever an identifier is added in an equivalence class the equivalence class's root is added in the *definers* set for the file containing the particular identifier. Identifiers are added in equivalence classes each time there is a semantic match with a previous instance of that identifier. By looking at all instances where identifiers appear in the C grammar and in an operational definition of the C preprocessor we can derive an exhaustive list of cases where identifiers can reappear, and thus trigger a semantic match. Specifically, identifiers can reappear as:

- part of a C or a preprocessor expression
- declarations for previously declared objects
- an aggregate member designator
- a *typedef* name appearing in a declaration specifier
- a tag, part of an aggregate name, aggregate key or enumeration name
- a tag, part of a C99 [20] initializer designator
- variable declarations in old-style [21] function formal arguments
- an identifier replaced by a macro
- a formal macro argument appearing in a macro body
- a redefined macro
- an argument to an *undef* preprocessor command
- labels or targets of *goto* statements

In each of the above cases, when an identifier is added in a non-empty equivalence class, the file $r \in F$ where the class's root appears and the file $f \in F$ containing the identifier are appropriately added in the corresponding *definers* relation.

$$R'_d(c, f) = R_d(c, f) \cup \{r\}$$

Note that files containing only data or isolated code statements are members of the *providers* relation R_p and do not participate in a *definers* relation R_d.

2.3. Reasoning About Included File Dependencies

The last step for determining the included files that are really required by a given compilation unit involves calculating the union of the transitive closure of the *providers* relation R_p with the transitive closure of the *definers* and *includers* relations R_d and R_i. For this we define a new relation

$$R_{di}(c, f) = R_d(c, f) \cup R_i(c, f) \tag{1}$$

and then calculate the set of files $I'(c)$ that a compilation unit compiled from c must include as

$$I'(c) = R_{di}(c, c)^+ \cup \bigcup_{\forall p \in R_p(c)} R_{di}(c, p)^+ \tag{2}$$

What the above formulation expresses is that a file $i \in F$ is required, that is $i \in I'(c)$, if it

- contains a definition for an identifier;
- includes another file that is required; or
- provides code or data to the compilation unit c.

Files processed during the compilation, that are not marked as required, and are directly included by the compilation unit can have their corresponding *include* directives safely removed.

Our mathematical formulation can be expressed in code by means of the recursively defined function *mark_required*. This function takes as its single argument an identifier for a file being required for processing a given compilation unit. Associated with each file is a flag identifying whether this file is marked as "required". This is used for determining the set of required files and for avoiding endless recursion. The function *mark_required* is defined in pseudocode as follows:

```
mark_required(c, f)
{
        if (f.marked)
                return;
        f.marked = true;
        for (i in R_d(c, f))
                mark_required(c, i);
        for (i in R_i(c, f))
                mark_required(c, i);
}
```

The recursive algorithm is invoked through the function *required* with the file identifier of each compilation unit c as its argument.

```
required(c)
{
        mark_required(c, c);
        for (i in R_p(c))
                mark_required(c, i);
}
```

Listing 1. Example code.

```
/* sys/types.h */
typedef unsigned long ino_t;

/* sys/stat.h */
struct stat {
        short    st_dev;        /* inode's device */
        ino_t    st_ino;        /* inode's number */
};

/* stdlib.h */
void exit(int);

/* string.h */
char *strcpy(char * restrict, const char * restrict);

/* cdefs.h */
#include <copyright.h>

/* copyright.h */
static char copyright[] = "Copyright 2007 A. Holder";

/* main.c */
#include <cdefs.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>

main(int argc, char *argv[])
{
        struct stat buff;
        exit(!(argc == 2 && stat(argv[1], &buff) == 0));
}
```

The algorithm's implementation demonstrates that, after parsing and semantic analysis, the set $I'(c)$ for a compilation unit c can be calculated in no more than $|I'(c)| + |R_p(c)|$ operations.

2.4. Example

Consider the code example shown in Listing 1. Processing `main.c` as a compilation unit c will establish the following relations.

$$R_p(c) = \{\texttt{copyright.h}\} \quad (3)$$
$$R_i(c, \texttt{sys/types.h}) = \{\texttt{main.c}\} \quad (4)$$
$$R_i(c, \texttt{sys/stat.h}) = \{\texttt{main.c}\} \quad (5)$$
$$R_i(c, \texttt{stdlib.h}) = \{\texttt{main.c}\} \quad (6)$$
$$R_i(c, \texttt{string.h}) = \{\texttt{main.c}\} \quad (7)$$
$$R_i(c, \texttt{cdefs.h}) = \{\texttt{main.c}\} \quad (8)$$
$$R_i(c, \texttt{copyright.h}) = \{\texttt{cdefs.h}\} \quad (9)$$
$$R_d(c, \texttt{sys/stat.h}) = \{\texttt{sys/types.h}\} \quad (10)$$
$$R_d(c, \texttt{main.c}) = \{\texttt{sys/stat.h}, \texttt{stdlib.h}\} \quad (11)$$

Most of the above relations are trivially established. Equation 10 holds, because the type definition `ino_t` is defined in `sys/types.h` and referred by `sys/stab.h`. Similarly, equation 11 holds, because the `stat` structure tag and the `exit` function referred by `main.c` are defined correspondingly in `sys/stat.h` and `stdlib.h`.

Let us now apply the algorithm we described in Section 2.3. In our particular example $\forall x : R_{di}(c, x) = R_d(c, x) \vee R_{di}(c, x) = R_i(c, x)$, so we do not need to elaborate equation 1. By calculating depth-first the transitive closure $R_{di}(c, \texttt{main.c})^+$ through equations 11, 5, 10, and 6 we obtain the left-hand side term of the union in equation 2.

$$R_{di}(c, \texttt{main.c})^+ = \{\texttt{sys/stat.h}, \texttt{stdlib.h}, \texttt{main.c}, \texttt{sys/types.h}\} \quad (12)$$

The corresponding right-hand side is established through equations 3 and 9 as

$$\bigcup_{\forall p \in R_p(c)} R_{di}(c, p)^+ = \{\texttt{cdefs.h}\} \quad (13)$$

and therefore through equations 2, 12, and 13 we obtain

$$I'(c)^+ = \{\texttt{sys/stat.h}, \texttt{stdlib.h}, \texttt{main.c}, \texttt{sys/types.h}, \texttt{cdefs.h}\}$$

Consequently, the directive including file `string.h` is not required, and can be safely removed. Furthermore, the result can be justified intuitively as follows:

- `sys/stat.h` is required for the `stat` tag used in `main.c`
- `stdlib.h` is required to define the `exit` function used in `main.c`
- `sys/types.h` is required to define the `ino_t` type definition, which is required by `sys/stat.h`, which is required by `main.c`
- `cdefs.h` is required, because it includes `copyright.h`, which provides data to the compilation unit
- `string.h` is not required, because `strcpy` isn't used anywhere

2.5. Conditional Compilation

A complication arises when conditional compilation is used as a method for configuration control [22]. In such a case differently configured compilations of the same unit can result in different include file requirements. As an example, processing a source file with the macro *unix* defined might result in finding that the included header file *windows.h* is not required, but processing the same source file with *WIN32* defined might result in finding that the header file *unistd.h* is not required. This problem can be obviated by repeatedly processing the same compilation unit under many possible configurations. This is possible, because the equivalence classes we outlined in Section 2.1 apply to lexical tokens and can therefore be maintained across multiple passes on the same compilation unit. Similarly, we also maintain across the different runs the relations R_p, R_i, and R_d that we described in Section 2.2. After processing all configurations, the algorithm for finding the included files that are not required is applied on the cumulative contents of these relations.

From a practical perspective the problem of this method lies in determining the set of macro definitions that will cover the processing of a large percentage of the code base (ideally all code lines). The problem is often simplified because many projects provide a build configuration that accomplishes this task for the benefit of other static verification tools. In some cases a macro named *LINT*—after the tool of the same name [23]—is used for this purpose. Nevertheless, there can be configurations that are mutually incompatible; for instance those covering hardware architectures with different characteristics, or alternative implementations of the same functionality. In such cases our approach involves processing the files multiple times, each time with the macros controlling the configuration set to a different value. Such a setup increases the amount of code coverage with each configuration added at the expense of additional processing time and space. We demonstrate this increase in code coverage in Section 3.3.

In order to assist developers locating and specifying the appropriate configurations we can maintain for each file the number of lines that were skipped in all configurations by conditional compilation directives. With appropriate tool support, developers can issue a query to see which files contain the largest number of unprocessed lines, and then view a listing of each file with markings indicating the lines that were not processed. This allows developers to focus on low hanging fruit, adding macro definitions or configurations that will increase the code coverage and thereby the fidelity of the include file optimization process.

2.6. Error Reporting

Reporting unused included files by reference to a specific file and line number (as is typically done in compiler warning messages) can be implemented by maintaining another relation associated with each file containing, for every file it includes, the line numbers of the directives that include it (a file can be included more than one time). However, when dealing with multiple processing passes over the same file (the method we use to deal with conditional compilation and other configuration control techniques) the same *include* directive can include different files. In practice, this can occur either extra-linguistically when each pass is performed with a different include file path, or through preprocessor facilities—by defining different macro values for each configuration and using the corresponding macro as an argument for an *include* directive. The following example illustrates the latter case.

```
#if defined( __alpha__ )
#define FILE "alpha.h"
#elif defined( __i386__ )
#define FILE "i386.h"
#endif

#include FILE
```

An additional relation can be used to overcome this complication. Every *include* directive is associated with an *include site*. An include site identifies (by means of the file and the line number the corresponding directive appears) the position of an include directive that could be a target for removal. Each include site is associated with: the set of files included by its directive, and a boolean value indicating whether at least one of the included files was required by the compilation unit including it. A mapping from a compilation unit's line numbers to the corresponding include sites can then be used to update and locate the include sites that do not include even one used header. These located include sites are then reported as containing unused included files that can be safely removed.

3. Application

We integrated the algorithm described in the previous sections into the *CScout* source code analyzer and refactoring browser [9]. *CScout* can process workspaces of multiple projects (we define a project as a collection of C source files that are linked together) mapping the complexity introduced by the C preprocessor back into the original C source code files. *CScout* takes advantage of modern hardware advances (fast processors and large memory capacities) to analyze C source code beyond the level of detail and accuracy provided by current compilers and linkers. The analysis *CScout* performs takes into account the identifier scopes introduced by the C preprocessor and the C language proper scopes and namespaces.

3.1. Case Studies

Over the last few years we have performed a number of experiments and case studies to establish the magnitude of the needlessly included files problem and our approach's efficacy in addressing it.

The largest case study took place in 2007, where we tested our approach on 32 medium and large–sized open-source projects. These were: the Apache httpd 1.3.27, Lucent's awk as of Mar 14th, 2003, bash 3.1, CVS 1.11.22, Emacs 22.1, the kernel of FreeBSD HEAD branch as of September 9th, 2006 LINT configuration processed for the i386, AMD64, and SPARC64 architectures, gdb 6.7, Ghostscript 7.05, gnuplot 4.2.2, AT&T GraphViz 2.16, the default configuration of the Linux kernel 2.6.18.8-0.5 processed for the x86-64 (AMD64) architecture, the kernel of OpenSolaris as of August 8th, 2007 configured for the Sun4v Sun4u and SPARC architectures, the Microsoft Windows Research Kernel 1.2 processed for the i386 and AMD64 architectures, Perl 5.8.8, PostgreSQL 8.2.5, Xen 3.1.0, and the versions of the programs bind, ed, lex, mail, make, ntpd, nvi, pax, pppd, routed, sendmail, tcpdump, tcsh, window, xlint, and zsh distributed with FreeBSD 6.2. The FreeBSD programs were processed under FreeBSD 6.2 running on an i386 processor architecture, while the rest, where not specified, were

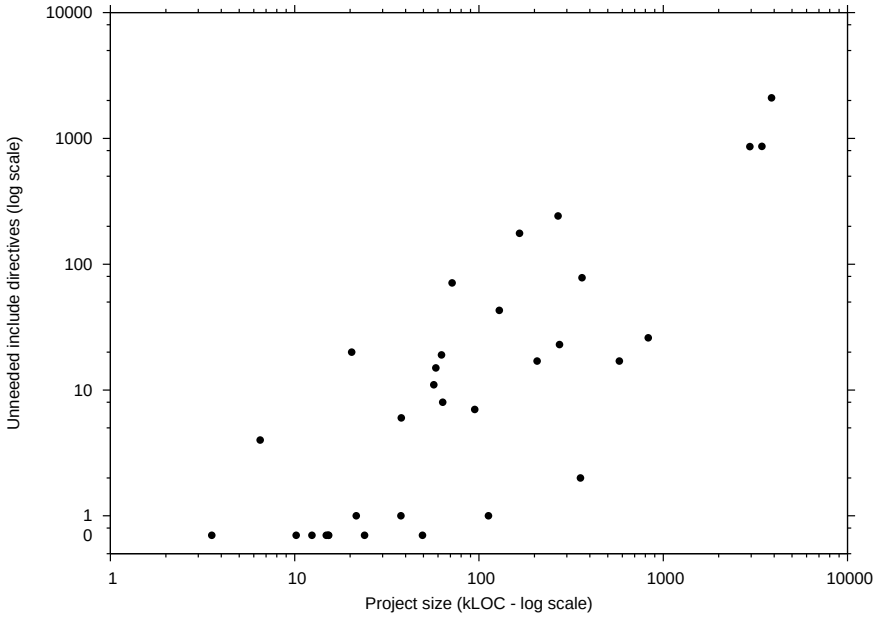

Figure 1. Unneeded include directives in projects of various sizes.

configured under openSUSE Linux 10.2 running on an AMD64 processor architecture. For expediency, we selected the projects by looking for representative, widely-used, large-scale systems that were written in C and could be compiled standalone. The processed source code size was 14.2 million lines of code.

A summary of the results appears in Figure 1. As we can see, unneeded header files are rarely a problem for projects smaller than 20 KLOC, but become a significant one as the project's size increases. (The chart's abscissa also includes a notional value of zero where projects without include directive problems are indicated.)

For the two largest systems, Linux and FreeBSD, we also verified that code resulting from removing the identified unneeded include directives could actually compile. We wrote a small script to remove those directives, and compiled the resulting source code without a problem. The compilation time of the corrected source code was not significantly reduced.

For the Linux case we also verified that the linked kernel generated after removing the extraneous header files was the same as the original one. The two generated kernel images, had the same size (9.6 MiB), but their contents were not identical. The reason was differences in timestamps embedded in object files. A subsequent comparison of the disassembled image files uncovered only a variation in a single assembly language instruction. This was traced to a one byte difference in the size of two compressed object files that were embedded in the kernel. Again, the source code of the two object files was identical; their difference stemmed from file timestamps located in an embedded *cpio* archive.

Moreover, for the FreeBSD case we looked at the possibility of integrating the changed source code in the system's production version. Specifically, in 2003, we applied *CScout* to the source code of the FreeBSD operating system kernel (version 5.1), in five separate architecture-specific configurations (i386, IA64, AMD64, Alpha, and SPARC64); the configurations of the five architectures were processed in a single run. In total we processed 4,310 files (2 MLOC) containing about 35,000 *include* directives. In the that set 2,781 directives were found as including files that were not being required. From the files that were found as needlessly included, 741 files included from 386 different sites could in practice not be removed, because the same site also included (probably under a different configuration) files that were identified as required. The remaining 2,040 *include* directives could be safely removed. The processing took 330 CPU minutes on a 1.8GHz AMD-64 machine and required 1.5GB of RAM. We published a list of the corresponding changes, and discussed with other FreeBSD developers the possibility of committing them to the system's source code repository. We backed off after developers pointed out that the files from which the include directives were automatically removed violated various style guidelines. Problems included consecutive blank lines and empty blocks of preprocessor conditionals. Thus, it turned out that, although our approach can identify unneeded include directives, their automatic removal is not entirely trivial.

Finally, we also applied our approach on proprietary production code. We processed an architectural CAD project that is under active development since 1989 [24, 25]. At the time the project consisted of 231 files (292,000 lines of code), containing 5,249 *include* directives. Following *CScout*'s analysis 765 *include* directives from 178 files were identified as superfluously included and were removed. The application was subsequently compiled, tested without a single problem, and is currently in production use by thousands of clients.

3.2. Discussion of the Results

The substantial number of unused header files in large systems requires some explanation. It could be attributed to the following reasons.

Removed code A developer removing code from a C file (or moving the code to another file) might fail to remove the corresponding header file include directives. This is to be expected, because a single include directive might serve multiple and different elements. It is difficult for a developer to know when the last reference to a header file has been removed from a C file.

Moved header file definitions A developer moving a definition from one header file to another would be hesitant to remove references to the first header file from all the C files it appears in. First, this would be a lot of work, as the header might appear in many C files, and, second, the header might also be required for other elements it defined.

Incomplete configurations As we are by no means acquainted with the sample programs we examined, our analysis of them might involve configurations in which large parts of the program functionality are not present. Header files reported as unused might in fact be required by code that is conditionally compiled in specific configurations.

Disjoined configurations Arguably, the problem of incomplete configurations should not occur in well-written code. If a given header is only required for a particular, conditionally-compiled,

configuration, then the header's include directive should also be compiled only under the corresponding configuration.

To examine the dynamics of header file addition and removal, we went over 100,324 records of modifications that were made to the FreeBSD kernel trunk over the last thirteen years (from 1994-05-24 to 2007-11-19). We examined differences between successive revisions, counting the number of code lines and include directives that were added or removed in each revision. We found that header include directives amounted to 1.50% of code lines added, but to 1.46% of code lines removed. Adjusting this difference of 0.04% would amount to deleting another 548 include directives; a figure roughly equal in magnitude to the extraneous header include directives we found. This finding may suggest that when removing code developers fail to remove the corresponding header include directives.

In addition, to examine the effect of incomplete and disjoined configurations we associated with each file containing unneeded include directives the number of lines that were not processed in that file due to conditional compilation directives. We could therefore establish the number of unneeded header include directives located in files that were fully processed. The results appear in Table I on the row titled "... in f.p. files". These numbers establish a lower bound on the number of headers that can be removed: more headers could be candidates for removal if processing additional configurations didn't reveal that these headers were required, after all. The numbers could also explain a part of the large number of unneeded include directives we found: about half of the unneeded headers may be an artifact of disjoined configurations.

3.3. Dealing with Multiple Configurations

In order to determine how processing the same files under multiple configurations increases the code coverage, we processed the FreeBSD kernel (HEAD as of 2006-09-18) under the seven different combinations of the tier-1 (fully supported) architectures: i386, AMD64, and SPARC64. The results appear in Figure 2. Each node indicates the configuration(s) that were processed, the corresponding total lines of code (in millions), and the code coverage as a percentage of the total code. The lowest amounts of code coverage occur in the case where a single architecture is processed. When two architectures are processed together (there are three such combinations in our example) the achieved code coverage is greater, even though the number of lines processed is higher than those processed for each of the two architectures. Finally, the greatest code coverage (appearing in the node at middle of the figure) is achieved when all three architectures are processed as a single configuration. Further increases in code coverage could be achieved by processing additional configurations.

4. Discussion and Possible Extensions

The application of the approach we have described appears to provide results that are both accurate and useful. Its requirements in terms of memory and CPU time may preclude its integration in a typical compile cycle, but easily allow its periodic application over a code base as a quality assurance measure.

Although the results obtained from this method are *sound* (removing the files reported as needlessly included will always result in a correct compilation for the specified configuration), they are not *complete*.

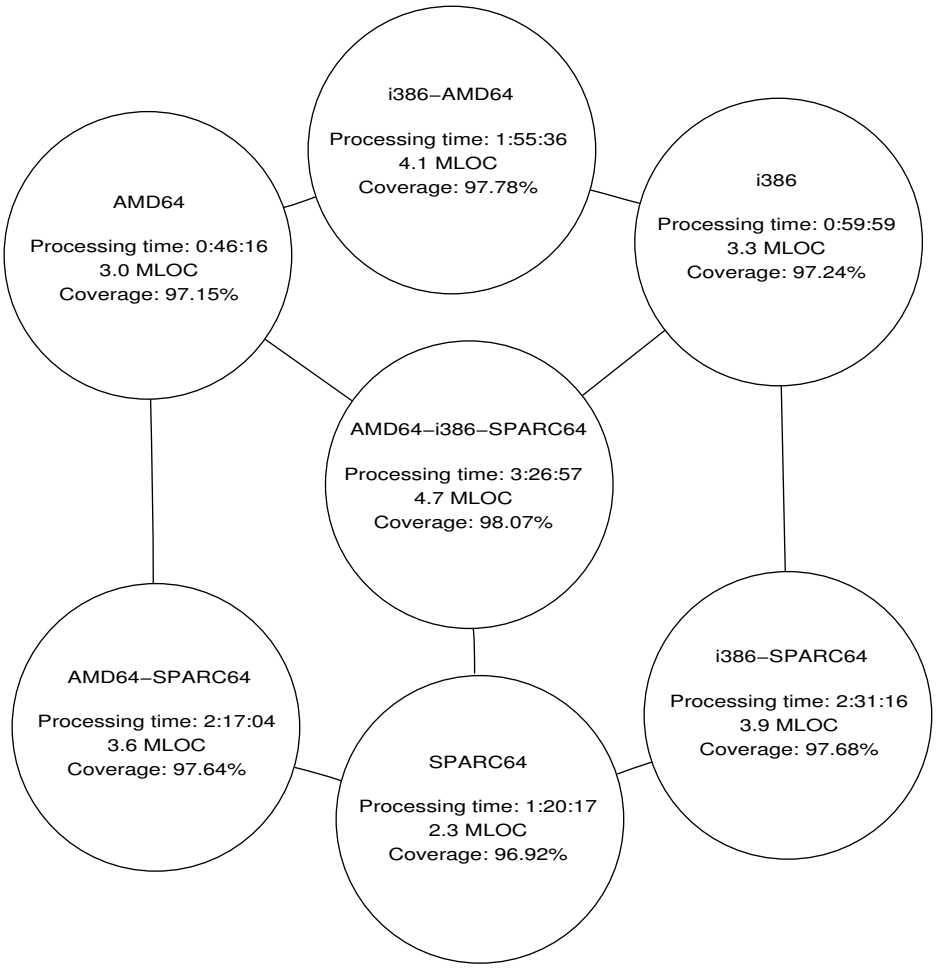

Figure 2. Processing more configurations yields increased code coverage.

First of all duplicate macro definitions or object declarations occurring in different included files will result in marking as required more files than are strictly necessary. This problem can be compounded when a forward declaration of a structure appears after its complete declaration. If at all subsequent points only the incomplete structure declaration is required, our approach will fail to detect this, and will mark as required the header file containing the complete structure declarations and all header files that the complete structure declaration requires.

Furthermore, our method will mark as required, through the R_p relation, header files containing code. In modern systems the trend is for header files to define static inline functions for elements

that in the past were defined through preprocessor macros, and rely on the compiler to optimize these functions away when they are not used. Such files will be marked as required through the R_p relation. The definition of the R_p relation could be amended to take into account only code and data that is actually exported by the compilation unit, but implementing this functionality is not trivial.

In addition, our method does not take into account different files, not present in the original set of included files, but part of the processed source code base, that if suitably included might result in an optimal (in e.g. terms of namespace pollution) set of included files. For instance, a source code base might have a small header file containing only incomplete structure declarations, and a larger header file containing the full structure declarations. If the small header is not included in a given compilation unit, our method will not suggest that the larger header file include directive can be removed.

Finally, our approach may fail in pathological cases where parts of a single syntactical element (for example a structure or function definition, or even a single statement) span various files. For instance, one could place each C keyword in a separate file (*while.h*, *if.h*, *else.h*, etc.) and include that file instead of writing the keyword. Fixing this requires changing the definition of the relations to include all the files from which the tokens comprising an element being defined come from, rather than just the token of the identifier being defined.

Although the approach we have described works for the traditional style of C code, modern compilers and style guidelines are moving toward a slightly different direction. The complexity of current C++ header files is changing the way header files are being used. Traditionally programmers were advised to avoid nesting header includes [26]. In contrast, modern style guidelines require each included file to be self-sufficient (compile on its own) by including all the requisite header files [27, p. 42]. Header files in turn should be protected against being processed multiple times by so-called *internal include guards*. That is, the contents of each header file are placed in a block like the following.

```
#ifndef FILE_H_INCLUDED
#define FILE_H_INCLUDED
...
#endif
```

Modern compilers can recognize this idiom and avoid even opening the file after processing it for the first time. Under such a style regime, our approach would have to be modified to treat each header file as a potentially standalone compilation unit. The relations we described in Section 2.2 would still be required, but the inference for deciding which files to include would have to be modified to match this style's guidelines.

Clearly, including the correct headers is far from trivial, and programmers need all the help tools can provide them.

Acknowledgements

Alexios Zavras gave insightful comments on earlier versions of this paper. Vasilis Kapouleas helped in the formalization of the algorithm's description. The FreeBSD community provided me with hardware resources for testing the effectiveness of the approach on version 5.1 of the kernel, while Ruslan Ermilov, Bruce Evans, John-Mark Gurney, Alexander Langer, M. Warner Losh, Juli Mallett, and Peter Wemm examined the proposed changes and commented on them. Markos Gogoulos setup and maintained the testing environment. Furthermore, I wish to thank Microsoft Corporation (and Fotis Draganidis in particular) for providing me access to the Windows

Research Kernel. Most importantly, the comments of this paper's anonymous referrees have contributed greatly to its improvement.

REFERENCES

1. Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, December 2002.
2. Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Technical Conference Proceedings*, Berkeley, CA, June 2002. USENIX Association.
3. American National Standard for Information Systems — programming language — C: ANSI X3.159–1989, December 1989. (Also ISO/IEC 9899:1990).
4. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1992.
5. William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.
6. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, 2000.
7. Jean-Marie Favre. Preprocessors from an abstract point of view. In *Proceedings of the International Conference on Software Maintenance ICSM '96*. IEEE Computer Society, 1996.
8. Greg J. Badros and David Notkin. A framework for preprocessor-aware C source code analyses. *Software: Practice & Experience*, 30(8):907–924, July 2000.
9. Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, November 2003.
10. Stuart I. Feldman. Make—a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, 1979.
11. Panos E. Livadas and David T. Small. Understanding code containing preprocessor constructs. In *IEEE Third Workshop on Program Comprehension*, pages 89–97, November 1994.
12. Marian Vittek. Refactoring browser with preprocessor. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 101. IEEE Computer Society, 2003.
13. Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL: a graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, 2006.
14. László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus schema for C/C++ preprocessing. In *CSMR '04: Proceedings of the Eighth European Conference on Software Maintenance and Reengineering*, pages 75–84. IEEE Computer Society, March 2004.
15. Yijun Yu, Homy Dayani-Fard, and John Mylopoulos. Removing false code dependencies to speedup software build processes. In *CASCON '03: Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 343–352. IBM Press, 2003.
16. Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 281–292, Washington, DC, USA, 2001. IEEE Computer Society.
17. Lerina Aversano, Massimiliano Di Penta, and Ira D. Baxter. Handling preprocessor-conditioned declarations. In *SCAM'02: Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 83–93, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
18. Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, pages 196–207, Washington, DC, USA, 2000. IEEE Computer Society.
19. Diomidis Spinellis. Code finessing. *Dr. Dobb's*, 31(11):58–63, November 2006.
20. International Organization for Standardization. *Programming Languages — C*. ISO, Geneva, Switzerland, 1999. ISO/IEC 9899:1999.
21. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, first edition, 1978.
22. Henry Spencer and Geoff Collyer. #ifdef considered harmful or portability experience with C news. In Rick Adams, editor, *Proceedings of the Summer 1992 USENIX Conference*, pages 185–198, Berkeley, CA, June 1992. USENIX Association.
23. Stephen C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, December 1977.
24. Diomidis Spinellis. Tekton: A program for the composition, design, and three-dimensional view of architectural subjects. In *4th Panhellenic Informatics Conference*, volume I, pages 361–372. Greek Computer Society, December 1993. In Greek.

25. Diomidis Spinellis. Reliable software implementation using domain specific languages. In G. I. Schuëller and P. Kafka, editors, *Proceedings ESREL '99 — The Tenth European Conference on Safety and Reliability*, pages 627–631, Rotterdam, September 1999. ESRA, VDI, TUM, A. A. Balkema.
26. L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, and Mark Brader. Recommended C style and coding standards. Available online `http://sunland.gsfc.nasa.gov/info/cstyle.html` (January 2006). Updated version of the Indian Hill C Style and Coding Standards paper.
27. Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison Wesley, 2004.

AUTHOR'S BIOGRAPHY

Diomidis Spinellis is an Associate Professor at the Department of Management Science and Technology at the Athens University of Economics and Business, Greece. His research interests include software engineering tools, computer security, and programming languages. He has written the two *Open Source Perspective* books: *Code Reading* (Software Development Productivity Award 2004), and *Code Quality* (Software Development Productivity Award 2007). He is a member of the *IEEE Software* editorial board, authoring the regular *Tools of the Trade* column. He is a FreeBSD committer and the author of a number of open-source software packages, libraries, and tools. He holds an MEng in Software Engineering and a PhD in Computer Science, both from Imperial College London. Dr. Spinellis is a senior member of the ACM, and a member of the IEEE, and the Usenix Association.

Id: include.tex 1.35 2008/02/29 21:09:07 dds Exp